Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels

Murad Qasaimeh*, Kristof Denolf†, Jack Lo†, Kees Vissers†, Joseph Zambreno*, and Phillip H. Jones*
*Iowa State University, IA, USA, †Xilinx Research Labs, CA, USA

Abstract—**Developing high performance embedded vision applications requires balancing run-time performance with energy constraints. Given the mix of hardware accelerators that exist for embedded computer vision (e.g. multi-core CPUs, GPUs, and FPGAs), and their associated vendor optimized vision libraries, it becomes a challenge for developers to navigate this fragmented solution space. To aid with determining which embedded platform is most suitable for their application, we conduct a comprehensive benchmark of the run-time performance and energy efficiency of a wide range of vision kernels. We discuss rationales for why a given underlying hardware architecture innately performs well or poorly based on the characteristics of a range of vision kernel categories. Specifically, our study is performed for three commonly used HW accelerators for embedded vision applications: ARM57 CPU, Jetson TX2 GPU and ZCU102 FPGA, using their vendor optimized vision libraries: OpenCV, VisionWorks and xfOpenCV. Our results show that the GPU achieves an energy/frame reduction ratio of 1.1–3.2× compared to the others for simple kernels. While for more complicated kernels and complete vision pipelines, the FPGA outperforms the others with energy/frame reduction ratios of 1.2–22.3×. It is also observed that the FPGA performs increasingly better as a vision application's pipeline complexity grows.**

Index Terms—**Embedded Vision, GPUs, FPGAs, CPUs, Energy Efficiency.**

I. Introduction

Image sensors are increasingly becoming an essential component of a wide range of embedded system applications, such as: smartphones, autonomous cars, drones. This trend is a driving force for the development of energy-efficient image processing solutions. Energy-efficient image processing is especially important for tightly energy-constrained real-time embedded systems, as often their limited communication power budget or communication capabilities preclude them from streaming images to more powerful computing entities.

Both industry and academia have explored the development of acceleration engines to help meet the needs of embedded vision applications. Three common types of such accelerators are benchmarked in this case study: multicore CPUs, Graphic Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs). Each of these accelerators take a different approach to accelerating embedded vision applications. Multi-core CPUs make use of SIMD instruction extensions, such as: the ARM NEON SIMD engine, Intel's family of SSE, and dedicated vision processing units (VPU), such as Myriad [1]. The multi-threading programming model has made GPUs highly popular in this domain. GPUs provide massively parallel execution resources and high memory bandwidth. However, their high performance comes at the cost of high power dissipation [2]. FPGAs offer opportunities for exploiting low-level fine-grained parallelism by customizing data paths to the requirements of a specific algorithm/application [3].

Embedded vision applications can exhibit vastly different performance characteristics depending on their underlying hardware accelerator platform [4]. This varying behavior fundamentally stems from differences in accelerator micro-architectures, middleware support, and programming styles. This mixture of factors makes choosing the best application-to-accelerator mapping a nontrivial task for embedded vision application developers. They must take into consideration metrics, such as expected runtime performance, energy-efficiency, and programmability. An additional challenge facing developers is partitioning vision pipelines into phases that can run on available accelerators in the most efficient and cost-effective manner. In order to clearly understand how different hardware architectures may impact the performance of vision kernels, we analyze the performance of such accelerators for different vision kernels. In this paper, we evaluate the performance of three popular HW accelerators for vision applications: the ARM57 CPU, Jetson TX2 GPU, and ZCU102 FPGA. We propose and evaluate an easily reproducible benchmarking approach that only uses publicly available vision libraries: OpenCV, Nvidia VisionWorks and xfOpenCV, without adding any special platform specific code. All benchmark code is available at: https://github.com/isu-rcl/cvBench.

Contributions. In this work, we benchmark the performance of standard vision kernels on low-power embedded platforms. The main contributions of this paper are: (1) Benchmark representative vision kernels and complete pipelines for the ARM57 CPU, Nvidia Jetson TX2 (GPU-accelerated) and Xilinx UltraScale (FPGA-accelerated), (2) Insight into the reasons behind the observed run-time, power, and energy consumption performance for each evaluated platform, and (3) An energy efficiency comparison between the three hardware accelerator platforms evaluated in terms of energy delay product (EDP).

Organization. The remainder of this paper is organized as follows. Section II reviews related work. Section III presents six categories of vision algorithms, and provides insights into the architectural differences between the hardware accelerators evaluated. In Section IV, we present the performance metrics used in this study and provide a detailed description of our measurement methodology. In Section V, we discuss our experimental results and observations. Finally, Section VI concludes the paper with outlooks for future work.

978-1-7281-2437-7/19/$31.00 ©2019 IEEE

II. Related Work

Most prior studies focus solely on comparing the performance of single vision kernels on embedded GPUs and FPGAs, with a few exceptions as discussed below. The comparison study in [5] analyzed the performance efficiency of FPGAs and GPUs on the GPU-friendly benchmark suite (Rodinia). They ported 15 of its kernels using Vivado HLS for the FPGA and OpenCL for host programs. The platforms used were a Virtex-7 FPGA and Tesla K40c GPU. Although this study includes some vision kernels such as: GICOV, Dilate, SRAD and MGVF, it was not mainly focused on benchmarking vision algorithms; it included other kernels for data mining, fluid dynamic, and physics simulation, etc [6].

Other comparison studies focused on a subset of vision kernels. For example, the study in [7] and [8] evaluated the performance of sliding window applications on FPGAs, GPUs and multi-core CPUs. They compared the performance of three applications: Sum of Absolute Differences (SAD), 2D convolution, and correntropy. The platforms used in their study were an Altera Stratix IV FPGA, an NVIDIA GeForce GTX 560, and an Intel Xeon Core i7. Another study in [9] focused on comparing the performance of morphological image filtering operations. The authors utilized the OpenCV library for a CPU and GPU (cv::CUDA module). For the FPGA platform, they used Vivado HLS video libraries and hand-optimized implementations. The platforms used in their study were the Zynq 7020 FPGA, Tegra K1, and Intel core i7. The work in [10] also focused only on applications such as normalized cross correlation and finite impulse response (FIR) filters. The study's evaluation included development time, component cost, and power consumption.

In our work, we evaluate the run-time performance and energy efficiency of different embedded hardware solutions over a wide range of standard vision kernels. We provide rationale for when and why specific hardware platforms perform well or poorly for specific vision kernels.

III. Background

In this section, we first present the characteristics of the hardware accelerators evaluated in this study. Then, we briefly discuss the three vision libraries widely used with these accelerators. Finally, we group vision kernels into categories based on their characteristics to understand the implications of the underlying hardware architectures on the performance of the kernels in their respective categories.

A. Embedded Platforms

1. Central Processing Unit (CPU): Modern CPUs are able to preform SIMD (Single Instruction, Multiple Data) instructions using multiple ALUs. These SIMD instruction sets are useful in the context of image processing, where operations are often repetitively applied to a continuous stream of data. This is particularly true in the context of computer vision, where most operations are performed over the entire image. Examples of SIMD architectures are: ARM NEON SIMD engine and Intel's streaming SIMD extensions (SSE).

2. Graphic Processing Unit (GPU):

As compared to general purpose CPUs, which have developed SIMD instruction extensions to help parallelize image processing type tasks, GPUs have taken the direction of evolving into a specialized SIMD architecture. This specialization has led to GPUs having simpler processing cores than high-performance general purpose CPUs. For example, they have simpler control logic, typically no branch prediction or prefetch, and small per-core memory. Simpler computing cores allow GPUs to pack many more cores into a chip than a general purpose CPU. GPU architectures perform extremely well on workloads that have little to no branching conditions or data dependences. Additionally, GPU architectures have specialized their memory architecture to support high-speed data streaming for image processing. For example, the L2 cache in the Jetson TX2 (Pascal GPU) is 2048 KB, which can fit a 1080p grayscale image.

3. Field Programmable Gate Array (FPGA): Instead of having a fixed processor-like design, FPGAs consist of an array of logic blocks, DSPs, on-chip BRAMs, I/O pads, and routing channels. In FPGA, custom data paths can be architected to stream pixels directly between computing units without needing to read/write from/to external memory. Moreover, the distributed on-chip BRAMs can be used to exploit data locality in vision kernels by keeping pixels on-chip (e.g Zynq UltraScale MPSoC FPGA has 32.1 Mb on-chip memory). With FPGAs, developers need to ensure that their customized designs meet timing and space requirements.

B. Computer Vision Libraries

A number of vision libraries have been optimized to target the hardware platforms discussed in the previous section. In this work, we focused on the most complete and commonly used libraries, as follows:

1. OpenCV: OpenCV is the de-facto standard C/C++ library for image and vision processing [11]. It is used by the computer vision community to create desktop and embedded vision applications. OpenCV has bindings for languages such as Python and Java. The latest version of OpenCV (at the time of writing this paper) is 4.0.

2. Nvidia VisionWorks: VisionWorks is a toolkit for computer vision and image processing released by Nvidia in 2015 [12]. It implements and extends the OpenVX standard, and is optimized for CUDA-capable GPUs. VisionWorks provides three programming models: immediate mode, graph mode and CUDA API. The latest version of VisionWorks is 1.6.

3. Xilinx xfOpenCV: The xfOpenCV library is a set of OpenCV functions optimized for Zynq and Zynq UltraScale devices by Xilinx [13]. It was first released in 2017, as part of the Xilinx reVISION stack. It has been implemented using HLS to work in their SDx development environment and provides a software interface for building vision pipelines on FPGAs. The latest version of the xfopenCV library is 2018.3.

C. Categories of Vision Kernels

Computer vision algorithms can be grouped into six categories based on their functionality. The complexity of these kernels grows over the first five categories. The last category includes composite kernels, which are composed of kernels from the other categories. The following discusses each category in more detail:

1. Input Processing: The kernels in this group are usually used as pre-processing steps. They include simple arithmetic operations to change the input format or number of channels into a desired format. Some examples of these kernels are: channel combine, channel extract, color conversion, and bit-depth conversion.

2. Image Arithmetic: Image arithmetic applies standard arithmetic/logic operations to one or more images. Because of the multi-dimensional nature of these pixel based operations, these kernels can benefit from highly parallel hardware architectures, such as GPUs and FPGAs. Furthermore, the data being processed is very localized; the algorithms can be distributed among different processing units without concerns of data dependencies. These operations include: thresholding, absolute difference, addition/subtraction, bitwise and/or/xor/not, multiplication, accumulate, accumulate squared, and accumulate weighted.

3. Image Filters: These algorithms compute the correlation between an input image and a kernel (small matrix of fixed-size). The data in these algorithms are local to the size of the kernel which is different from the arithmetic case where the operations were performed on a pixel basis. When the underlying hardware has enough local memory to accommodate the kernel size, the algorithm is still easily distributed among parallel processing units. On the other hand, nonlinear filters are more irregular as they have branching conditions. This impedes their decomposition into parallel blocks. These kernels include: filter2D, box filter, erode, dilate, median, pyramid up, and pyramid down.

4. Image Analysis: Analytic kernels are typically used to understand characteristics of an image, such as color distribution, mean, maximum and minimum pixel value, etc. Also, they are usually placed at the end of vision pipelines to reduce the image into a decision variable (min/max locations). These kernels are filled with branching conditions and complex memory access patterns that negatively impact their performance on CPUs and GPUs. These operations include: histogram, mean/std, min/max location, table lookup, histogram equalization, and integral image.

5. Geometric Transformation: Transformations in geometric space are essential to understanding the 3D world through the lens of a 2D image sensor. These kernels include matrix multiplication that map effectively into highly parallel architectures composed of simple computing blocks (e.g. GPU). While these kernels are simple, their performance is negatively affected by irregular memory access patterns. These kernels include: remap, resize, affine warp, and perspective warp.

6. Composite Kernels: The kernels in this category are composed in part of kernels from the previously described categories. Examples of these composite kernels are: feature extraction, stereo block matching, and optical flow. *Feature extraction* is used to find interesting pixels in an image. Once features are extracted, they are no longer stored as a continuous block of adjacent pixels in memory. This forces other kernels to load non-continuous memory addresses, which may hinder parallelism performance. *Stereo block matching* uses two cameras, with known position and characteristics, to compute disparity by comparing overlapped regions, leading to a high computational load. *Optical flow* is used to estimate the apparent motion of objects between two consecutive images. Optical flow can be computed for each pixel (dense) or a subset of pixels (sparse).

IV. Experimental Methodology

This section describes the performance metrics and measurement techniques used, and introduces our study's benchmarking approach.

A. Performance Metrics

Selecting proper metrics is essential for assessing the energy efficiency of vision kernels running on different hardware accelerators. These metrics should provide meaningful interpretation and a fair way for comparison. In this subsection, we discuss the evaluation metrics used in our study:

Run-time: Run-time performance of vision kernels can be evaluated by measuring the elapsed time (delay time) between the start and end of a kernel's code. We use a high resolution timer to accurately measure elapsed time while executing on HW accelerators. In our study, we measured the execution time only, and excluded the time required for copying images from/to the external memory of CPUs and GPUs, and the time for configuring datamovers in FPGAs.

Energy: Energy consumption per frame quantifies the amount of electrical energy dissipated by hardware accelerators to perform a kernel's operations on one frame. It is measured as the power consumed during the delay time to process a frame. Device power can be divided in two parts: (1) Static power: represents the amount of power consumed when no active computation is taking place (system is idle), (2) Dynamic power: represents the amount of power consumed above the static power level when the system is computing.

Energy-Delay Product (EDP): Run-time or energy per frame alone do not show the entire picture. A hardware platform can be extremely low power while being too slow to be of practical use. The Energy Delay Product (EDP) [14] metric takes into account the throughput of the algorithm measured in (ms/frame) along with the energy consumed per frame (mJ/frame). EDP is the product of energy/frame and delay time. This way, a fair comparison can be made when deciding which hardware architecture is better suited for specific computation. Lower EDP is better which means that the hardware architecture can finish specific computation tasks using less power in less time.

B. Measurement Techniques and Platforms:

In this study, we evaluated two popular platforms for deploying embedded vision applications: Nvidia Jetson TX2 and Xilinx ZCU102. These platforms come equipped with an on-board power measuring IC that can measure multiple power rails such as: CPU cores and GPU cores on the Jetson, and programmable logic, full power CPU cores and low power CPU cores on the FPGA platform. On the Jetson TX2, shell scripts (running on its ARM CPU) sample power rails and log their values along with the system's timestamp into text files. The act of measuring power consumes power, thus consequently affects the results. The presented data in this paper has been corrected for this. On the ZCU102, the Xilinx system controller tool (running on a PC) communicates over UART with a separate microcontroller (MSP430) that is controlling and reporting power data (so no correction needed).

For every benchmark, we first processed 1000 frames on the CPU core of the platform and then 1000 frames on the hardware accelerated part of the platform. This can be seen in Figure (1), where the first two vertical lines mark the first 1000 frames on the CPU and the following two lines mark the last 1000 frames on the hardware. We computed the average frame rate by measuring the time between vertical lines and divided it by 1000. All frames were gray-scale with 1080p resolution. The x-axis represents the number of power samples taken for each platform. Note that the ZCU102 has a different sampling rate than the TX2.

Hardware environments. FPGA board: the Xilinx Zynq UltraScale+ MPSoC ZCU102 board has a 16nm XCZU9EG FPGA, and an on-board 4GB 64bit DDR4 RAM with a peak bandwidth of 136Gb/s. GPU board: the Nvidia Jetson TX2 (Pascal 256 CUDA cores (16nm)) has 8GBs of 128bit DDR4 RAM with a peak bandwidth of 477.6 Gb/s. Both the FPGA and GPU have on-chip ARM CPU cores with NEON SIMD optimization. The FPGA was clocked at 300 MHz, the ARM-A57 at 1.7 GHz, and the GPU at 998.4 MHz.

Software environments. We used three publicly available vision libraries: (1) OpenCV 3.4 (2) Nvidia's VisionWorks 1.6 and (3) Xilinx's xfOpenCV 2018.3. While the OpenCV code base already comes with some GPU accelerated code, it does not come with FPGA support. For this purpose, we used OpenCV compatible C++ wrappers for xfOpenCV kernels [15]. With this wrapped functionality we were able to compile the same OpenCV code for both GPU and FGPA. Both OpenCV and VisionWorks support full IEEE floating-point precision, while xfOpenCV supports 8 bit precision.

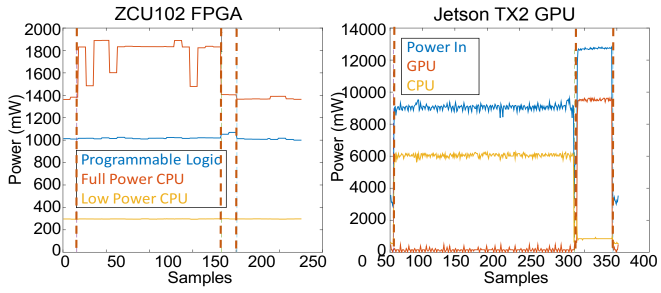

Fig. 1: Measuring power samples on the platform's CPU cores (first 1000 frames), and its FPGA or GPU (second 1000 frames).

C. Benchmarking Approach

In this study, we intentionally focused on evaluating the performance of out-of-the-box kernels from publicly available libraries (without writing special platform specific code around kernel calls) to give a fair comparison in terms of development efforts. For this reason, we first ran single kernel calls from OpenCV and VisionWorks libraries on the CPU and GPU, respectively, and instantiated a single kernel from xfOpenCV in FPGA fabric (even though small kernels utilize few FPGA resources). We then measured the efficiency of representative vision pipelines on the three HW accelerators to quantify their speed and energy efficiency on these more complete vision applications.

For single kernel evaluation, we compared the efficiency of the HW accelerators in terms of their energy consumption per frame. We measured a vision kernel's dynamic power while excluding the static power required to power the rest of the platform. This better reflects the actual workload that is being deployed to the system since certainly for small kernels, the *compute energy* [3] (energy consumed for computation only) and data transfer energy are usually dominated by the static power. In the vision pipeline evaluation, we compared the performance of HW accelerators in terms of their energy delay products (EDP). We used the total power consumption (static + dynamic), because it represents the actual power consumption when a complete system is deployed. We also measured the maximum frame rate achieved on the three HW accelerators. The theoretical frame rate on the FPGA is fixed for kernels that perform a single pass over the input image. Equation (1) shows an FPGA's frame rate when it is clocked at 300MHz for 1080p images.

$$FPS = \frac{300MHz}{1080 \times 1920 \times 1pixel/cycle} = 144 \qquad (1)$$

In order to have a sense of the amount of energy consumed for computation only, we measured the energy consumption of data movers in the FPGA and GPU. We implemented passthrough kernels which copy an image's pixels from one memory location to another without applying any arithmetic/logical operations. In the FPGA implementation, Xilinx's SDx tool instantiates data movers [16] for each input or output port to transfer data between the memory mapped domain and the stream domain. Table I shows that FPGA takes 6.945 ms to copy an entire image (1080p) with 0.41 mJ/frame, while GPU takes 1.298 ms with 0.19 mJ/frame. These values can be used to give a sense of the ratio of energy consumed for computation to data transfer in each kernel.

TABLE I: Data Movers Energy Consumption Measurements

Platform	Time/frame (ms)	Energy/frame (mJ/f)
FPGA	6.945	0.41
GPU	1.298	0.19

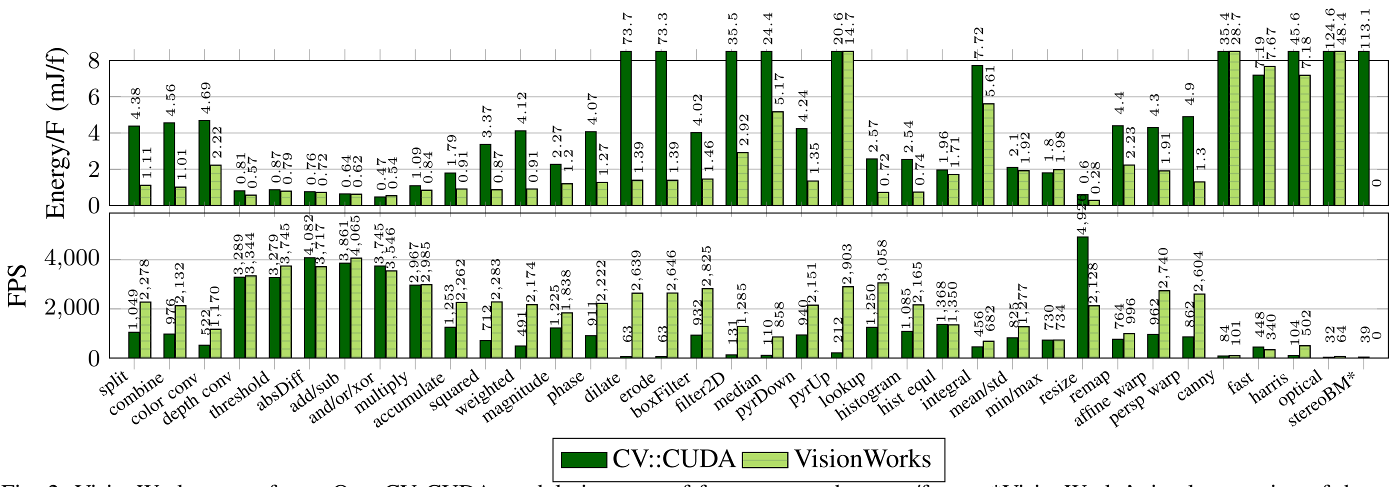

Fig. 2: VisionWorks outperforms OpenCV CUDA module in terms of frame rate and energy/frame. *VisionWorks's implementation of the stereoBM kernel is not publicly available.

V. Experimental Results

This section first presents the benchmarking results of single kernels from the six categories discussed in Section III. Then, a set of representative vision pipelines are evaluated.

A. Single Kernel Performance:

Before evaluating the run-time performance and energy/frame consumption of single kernels on the HW accelerators, we first compare two available GPU implementations: OpenCV CUDA module and Nvidia's VisionWorks toolkit. The OpenCV GPU module is written using CUDA and as a result benefits from the CUDA ecosystem. The Visionworks library applies many optimization techniques to boost performance, such as buffer reuse, kernel fusion, efficient use of streaming and CUDA textures, automatic scheduling across processing units, tiling and pipelining vision functions at the sub-frame level. Figure (2) shows the frame rate (bottom) and energy per frame (top) achieved by running vision kernels on the Jetson TX2. The Dark color represents OpenCV CUDA module, and the light color represents VisionWorks. We can observe that the VisionWorks implementation outperforms the OpenCV module in frame rate over all kernels. It achieved up to a 9.7× speedup compared to the OpenCV module. It also consumes less energy per frame over all kernels. It achieved up to a 6.3× reduction in energy consumption per frame. For this reason, in the rest of the paper, we will use only the VisionWorks implementation for the GPU.

Next, we measured the energy per frame consumption of vision kernels from the following six categories: (1) input processing, (2) arithmetic operations, (3) filter operations, (4) image analysis, (5) geometric transformation, and (6) composite kernels.

Input processing: The energy/frame of input processing kernels is shown in Figure (3). These kernels mapped well to the GPU and FPGA compared to the CPU because of their significant data parallelism, low complexity, and no data dependency. The GPU and FPGA achieved an average reduction ratio of 1.79× and 1.41× in energy/frame compared to the CPU. It also shows that GPU's implementation of bit-depth conversion achieved a 2.4× reduction compared to FPGA, because of the efficient use of streaming and CUDA textures in the VisionWorks kernel's implementation.

Image Arithmetic: The performance of arithmetic/logic operations is shown in Figure (4). It shows that simple operations such as: threshold, absDiff, add/sub, and bitwise and/or/xor can be efficiently implemented by the CPU. However, the CPU starts to perform poorly in kernels with multiplication operations, such as: multiply, accumulate squared, weighted, magnitude and phase. The GPU has the lowest energy/frame compared to the CPU and FPGA. The GPU's implementations achieved an average reduction ratio in energy/frame of 4.6× and 7.2× compared to CPU and FPGA, respectively. An expected result, as these algorithms can be granulated into many pieces that execute the same operation (SIMT).

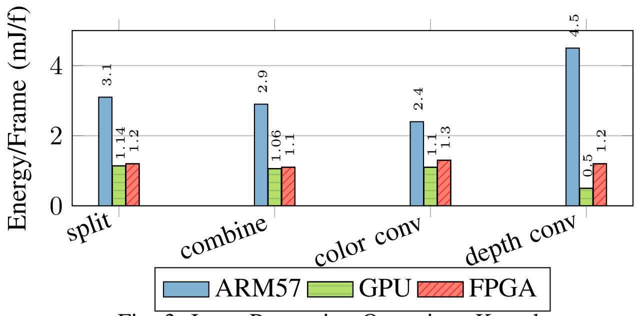

Fig. 3: Input Processing Operations Kernels

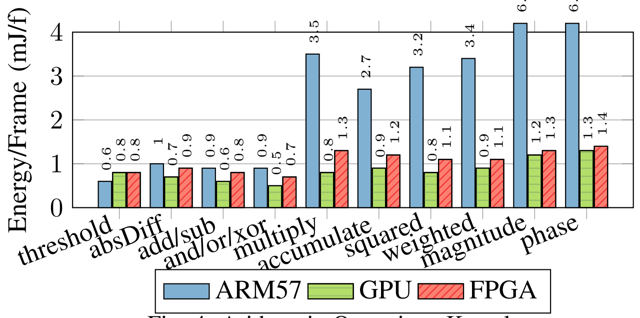

Fig. 4: Arithmetic Operations Kernels

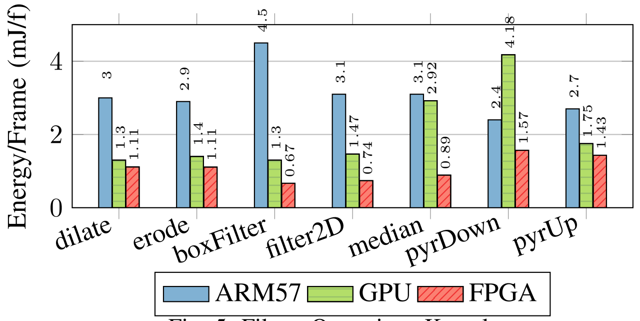

Fig. 5: Filters Operations Kernels

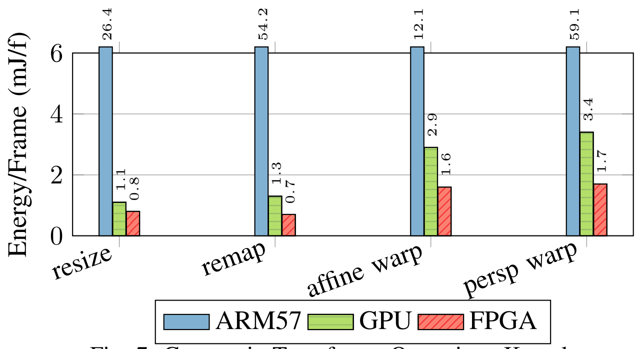

Fig. 7: Geometric Transforms Operations Kernels

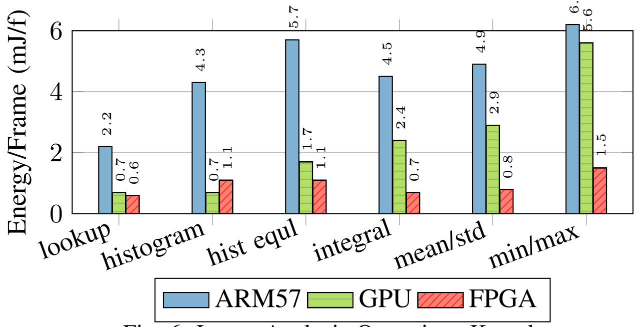

Fig. 6: Image Analysis Operations Kernels

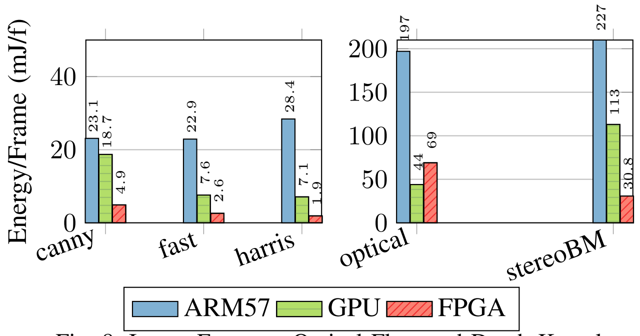

Fig. 8: Image Features, Optical Flow and Depth Kernels

Image Filters: In Figure (5), the results of filtering operations show that the FPGA performs better than the GPU and CPU for these kernels. The FPGA's implementation achieved an average reduction ratio of 1.8× and 7.4× in energy/frame compared to the GPU and CPU, respectively. The memory access patterns and mathematical complexity of linear filters (filter2D, box filter, pyramid up and pyramid down) maps well to the parallel processing of the GPU and FPGA. Median filters, however, are unlike linear filters. They do not use sequential data access and multiply-and-accumulate operations, but sort input elements and select the median of them, which makes them less straightforward to implement efficiently on a GPU. The morphological operations (dilate and erode) use hit and miss functions over a structuring element. These functions are more difficult to implement than filtering functions due to comparison and branching. This explains the low frame rate (as shown in Figure 2) and high energy/frame consumption of VisionWorks's implementations of small (3×3) filter kernels.

Image Analysis: The results of the image analysis kernels are shown in Figure (6). For kernels such as lookup table, histogram, and histogram equalization, the energy/frame consumption of the FPGA achieves an average reduction of 1.2× compared to the GPU. While for kernels with more branching conditions and complex memory access patterns, such as integral image, mean/std, and min/max locations, the FPGA's implementation achieved an average reduction ratio of 3.5× compared to the GPU.

Geometric Transformation: The results of the geometric transformation kernels are shown in Figure (7). The CPU performs poorly for these kind of operations compared to the GPU and FPGA. Also, the FPGA was more energy efficient compared to the GPU. It achieved a reduction of 1.6× in energy/frame for the resize and remap kernels, and 2× for affine warp and perspective warp kernels. The computations in the warp operations are more complex compared to resize and remap as mapping addresses need to be generated from 2×3 or 3×3 matrices before starting the mapping operation.

Composite Kernels: The last category in our study includes kernels for: (1) detecting image features (canny, fast and harris), (2) computing optical flow, and (3) computing disparity using stereo block matching. Figure (8) shows that the FPGA implementation of feature extraction kernels (canny, fast and harris) were more energy-efficient compared to the CPU and GPU by an average reduction of 7.7× and 3.5×, respectively. The steps to calculate sparse optical flow using the pyramid Lucas-Kanade algorithm includes extracting feature points from one frame and tracking them in the next frame. The FPGA implementation was able to detect 488 Harris corners compared to 94 for VisionWorks for the same input frame and parameters. Also, it was able to keep track of these points in the next frame. This explains the high energy/frame consumption in the FPGA implementation. Moreover, the VisionWorks's implementations of StereoBM is not open sourced yet, so the number reported in this paper is for the GPU implementation using OpenCV's CUDA module instead.

The average energy/frame reduction for the GPU and FPGA is shown in Table II. The ratio is with respect to CPU consumption (higher is better). We can observe a trend from simple kernels (top) to more complex kernels (bottom). The trend demonstrates that the performance of the GPU and FPGA compared to the CPU improves as kernels' complexity increases. For simple kernels (input processing and image arithmetic), the GPU shows the highest performance/energy efficiency, while for more complicated kernels (image filters,

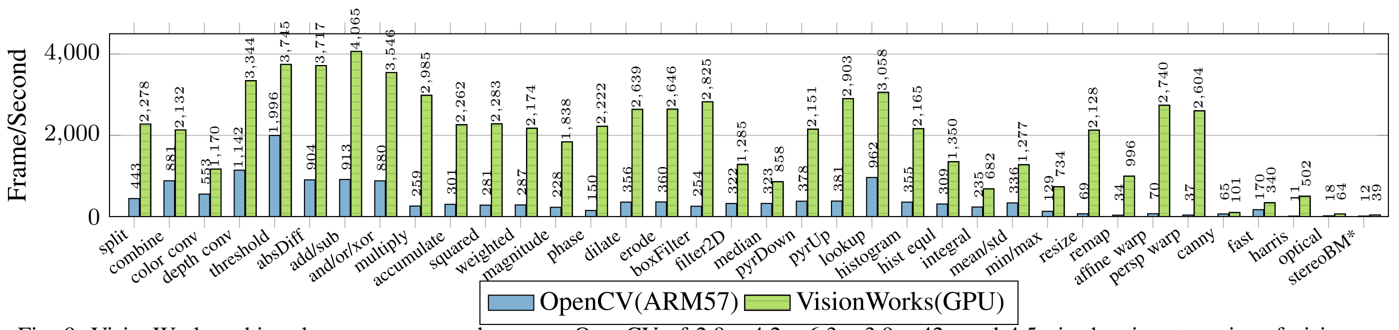

Fig. 9: VisionWorks achieved an average speed up over OpenCV of 2.9x, 4.2x, 6.3x, 3.9x, 42x and 4.5x in the six categories of vision kernels. *OpenCV's CUDA implementation of stereoBM kernel is used (VisionWorks is not publicly available).

image analysis and geometric transform), the FPGA shows the highest performance/energy efficiency. Moreover, as the complexity of kernels increase, the FPGA shows higher energy-efficiency compared to the GPU and CPU. This occurs due to the fact that more complex algorithms naturally occupy more resources on the programmable logic, as well as the fact that GPUs do not scale well for problems that are not easily divisible (data locality) or have many conditions or complex memory access patterns.

TABLE II: Ratios of Energy/Frame Reduction (Reference CPU)

	CPU	GPU	FPGA
Input Processing	1	**1.79**×	1.41×
Image Arithmetic	1	**3.19**×	2.93×
Image Filters	1	3.17×	**3.89**×
Image Analysis	1	2.34×	**5.67**×
Geometric Transform	1	10.3×	**16.6**×
Features/ OF/ StereoBM	1	7.44×	**22.3**×

For completeness, Figure (9) includes a frame rate comparison between the ARM57 CPU OpenCV and GPU VisionWorks implementations. It shows that VisionWorks implementations achieved an average speed up over OpenCV implementation of 2.9×, 4.2×, 6.3×, 3.9×, 42× and 4.5× for the six categories of vision kernels: input processing, arithmetic operations, filter operations, image analysis, geometric transformation, image features, optical flow, and stereo block matching. The FPGA's frame rate met the theoretical rate of Equation (1) for kernels performing a single pass over the input image. The theoretical frame rate for the FPGA is 144 fps when it is clocked at 300MHz for 1080p resolution images.

B. Complete Vision Pipeline Performance:

In this section, we evaluated the performance of the HW accelerators for four representative pipelines. Common steps in many computer vision pipelines include: pre-processing, feature extraction, and post-processing. The pipelines used in our study follow this structure: (1) background subtraction, (2) color segmentation, (3) stereo block matching, and (4) Harris corner tracking. These pipelines are implemented on the GPU using VisionWorks OpenVX graph mode to enable its advanced optimization techniques (buffer reuse, kernel fusion, etc.). We also pipelined the execution of kernels on the FPGA at pixel/frame level using xfOpenCV modules. In this way, the FPGA can leverage the fact that image pixels stays within the programmable fabric and avoids going back and forth to read/write from external memory. The pipelines evaluated in this paper are:

1. Background Subtraction: The background subtraction pipeline is used to detect changes in image sequences [17]. It is mainly used when regions of interest in images are foreground objects. The pipeline components include: subtraction, Gaussian filtering, threshold, erode and dilate, as shown in Figure (10).

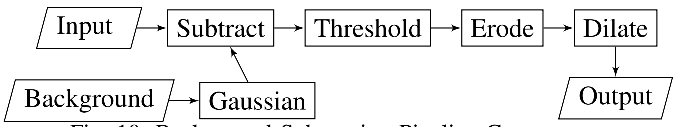

Fig. 10: Background Subtraction Pipeline Components

2. Color Segmentation: This pipeline is used to partition an image into multiple segments based on a specific range of colors. It converts the color format from RGB to HSV, then applies range thresholding to its three channels, and applies erode and dilate operations, as shown in Figure (11).

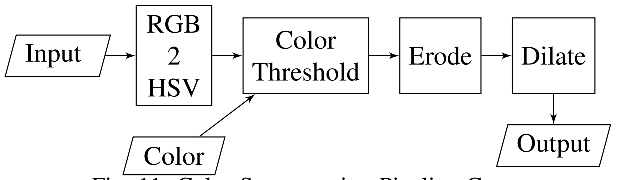

Fig. 11: Color Segmentation Pipeline Components

3. Harris Corners Tracking: This pipeline is used to detect and track feature points in a set of successive frames of a video. It takes in the current and next frame as inputs. It computes Harris corners from the current frame and outputs a list of tracked corners in the next frame. The pipeline uses five kernels as shown in Figure (12).

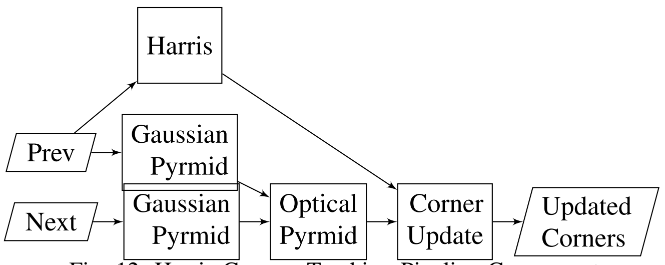

Fig. 12: Harris Corners Tracking Pipeline Components

4. Stereo Block Matching: This pipeline is used to generate a disparity map given the camera parameters and inputs from a stereo camera setup. It is used as a first step in creating a three dimensional map of an environment. The main components involved in the pipeline are shown in Figure (13). It consists of stereo rectification, remapping, and disparity estimation using a local block matching method.

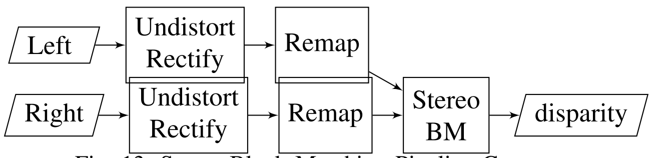

Fig. 13: Stereo Block Matching Pipeline Components

Figure (14) plots the Energy/frame and EDP comparison of the four pipelines, and shows the FPGA implementations consume less energy/frame compared to the CPU and GPU for all pipelines. The FPGA is also more efficient in terms of EDP (lower EDP is better). The FPGA's Energy/frame and EDP reduction ratio with respect to the GPU is listed in Table III. As the complexity of the pipeline grows, the energy/frame and EDP reduction ratio increases. More complex vision pipelines can use more of the FPGA programmable logic, reducing the relative impact of static power consumption. Additionally, data communicated between modules of the pipeline are kept on-chip in the streaming FPGA implementation.

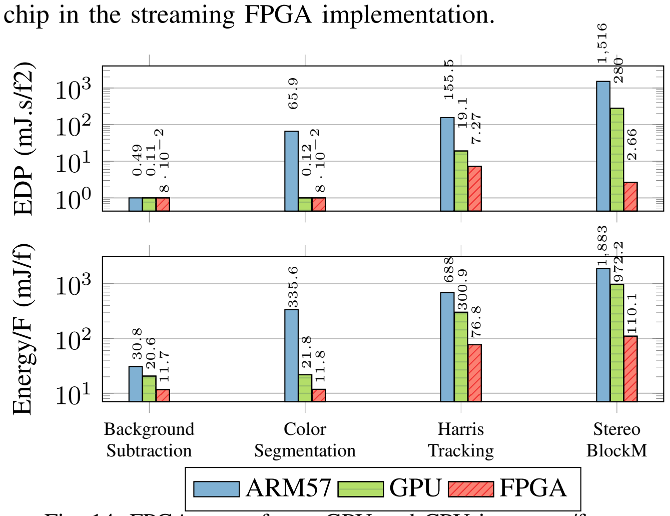

Fig. 14: FPGA outperforms GPU and CPU in energy/frame consumption and EDP

TABLE III: FPGA's Reduction Ratios with repsect to GPU

Pipeline	Energy/frame (mJ/f)	EDP (mJ.s/f2)
Background Subtraction	1.74×	1.32×
Color Segmentation	1.86×	1.41×
Harris Corners Tracking	3.94×	2.65×
Stereo Block Matching	8.83×	107.7×

VI. Conclusion

In this paper, we benchmarked algorithms from all the computer vision categories defined by the open standard, OpenVX, on both GPU- and FPGA-accelerated embedded platforms. We found that while many simple and easy-to-parallelize kernels perform well on GPUs (1.1–3.2× energy/frame reduction), for more complete vision pipelines, FPGAs outperform GPUs and CPUs (1.2–22.3× energy/frame reduction). Moreover, FPGAs perform increasingly better as the complexity of vision pipelines grow. This is evidenced by the energy-delay product, a metric that takes into account not only the energy/frame, but also algorithm throughput. Our future work will extend this analysis to the latest platform generation, like Nvidia's recently released AGX board, and will extend this benchmarking suite with key modules from machine learning and mixed pipelines composed of vision and machine learning kernels.

References

[1] M. H. Ionica and D. Gregg, "The movidius myriad architecture's potential for scientific computing," *IEEE Micro*, vol. 35, no. 1, pp. 6–14, 2015.

[2] S. Collange, D. Defour, and A. Tisserand, "Power consumption of gpus from a software perspective," in *International Conference on Computational Science*, pp. 914–923, Springer, 2009.

[3] H. Giefers, P. Staar, C. Bekas, and C. Hagleitner, "Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of gpu, xeon phi and fpga," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 46–56, IEEE, 2016.

[4] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *Application Specific Processors, 2008. SASP 2008. Symposium on*, pp. 101–107, IEEE, 2008.

[5] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of fpgas and gpus," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 93–96, IEEE, 2018.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, Ieee, 2009.

[7] P. Cooke, J. Fowers, G. Brown, and G. Stitt, "A tradeoff analysis of fpgas, gpus, and multicores for sliding-window applications," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 1, p. 2, 2015.

[8] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pp. 47–56, ACM, 2012.

[9] C. Brugger, L. Dal'Aqua, J. A. Varela, C. De Schryver, M. Sadri, N. Wehn, M. Klein, and M. Siegrist, "A quantitative cross-architecture study of morphological image processing on cpus, gpus, and fpgas," in *Computer Applications & Industrial Electronics (ISCAIE), 2015 IEEE Symposium on*, pp. 201–206, IEEE, 2015.

[10] E. Fykse, "Performance comparison of gpu, dsp and fpga implementations of image processing and computer vision algorithms in embedded systems," Master's thesis, Institutt for elektronikk og telekommunikasjon, 2013.

[11] G. Bradski, "Open source computer vision library." https://opencv.org/opencv-4-0-0.html, 2018.

[12] I. NVIDIA, "Nvidia visionworks toolkit." https://developer.nvidia.com/embedded/visionworks, 2018.

[13] I. Xilinx, "xfopencv library functions.." https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1233-xilinx-opencv-user-guide.pdf, 2018.

[14] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein, "Scaling, power, and the future of cmos," in *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, IEEE, 2005.

[15] "Computer vision overlays on pynq." https://github.com/Xilinx/PYNQ-ComputerVision. Accessed: 2018-12-17.

[16] V. Boppana, S. Ahmad, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig, "Ultrascale+ mpsoc and fpga families," in *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pp. 1–37, IEEE, 2015.

[17] "Background subtraction." https://docs.opencv.org/3.4/db/d5c/tutorial_py_bg_subtraction.html. Accessed: 2018-12-17.